Coding Standards for the NREL Commercial Buildings Group

Kyle S. Benne, Luigi Gentile Polese, Brent T. Griffith, Elaine T. Hale, Daniel L. Macumber, Nicholas L. Long

National Renewable Energy Laboratory

December 8, 2010

Contents

1 Introduction

The purpose of NREL's coding standard is to manage the complexity of the programming languages used in OpenStudio development while maintaining access to powerful features. The coding standard will help existing developers, new team members, and subcontractors work together in a cohesive team to create high quality software. Requiring the use of certain language features while prohibiting or restricting the use of others is one aspect of a coding standard. The other aspect is enforcing common idioms, styles, and conventions to help developers understand each other's code.

This coding standards documentation provides a starting point and reference manual for OpenStudio development. Developers should also become familiar with the continuous integration servers that run unit tests, dynamic memory leak analysis, and code coverage analysis. The website http://openstudio.nrel.gov/ will be maintained as a landing page from which developers can access development environment set up instructions; the project dashboard, svn repository, and trac; as well as updated versions of important documentation, including this document.

2 C++ Coding Standards

2.1 General Guidelines

These guidelines are not hard and fast rules, but rather general suggestions to keep in mind when making design decisions.

- Aim for extensibility and reuse. Put code in libraries as opposed to GUIs, break functionality into reusable pieces instead of large single-purpose chunks.
- Branch the svn repository if you need to make commits that will break the dashboard. Merge back as soon as possible.
- Simplify public interfaces. Outward facing interfaces should not expose templates or advanced programming concepts unless absolutely necessary. The public C++ interface should be as similar as possible to the SWIG interfaces. Remember that the target audience for OpenStudio (users and developers, C++ and SWIG targets) includes engineers with little programming experience.
- Implement generic interfaces first to allow access to the largest amount of functionality. Provide specialized interfaces as needed later.
- Do not use advanced concepts unnecessarily, e.g. "pImpl judiciously". For example, objects which need to be shared by many other objects should be implemented with the pImpl idiom, objects which do not need features offered by pImpl should not use it.
- Write unit tests that outline the task first. This will help with scope creep, and keep our collection of unit tests up to date.
- Use const to signify that a given method is not designed to modify a given class or input parameter, but do not spend a lot of time on const correctness. Instead use a limited interface to assure that object state remains valid.
- Especially when working on library code, make sure your objects and functions show up in the Doxygen and RDoc documentation.

2.2 External References

In addition to these standards we recommend the following sources of information as high quality external references for programming techniques and questions:

2.2.1 References

Meyers, S. (May 2005). *Effective C++: 55 Specific Ways to Improve Your Programs and Designs.* 3rd ed. Addison-Wesley Professional.

Sutter, H. (Nov. 1999). *Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions.* illustrated edition ed. Addison-Wesley Professional.

Sutter, H.; Alexandrescu, A. (Nov. 2004). *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices.* Addison-Wesley Professional.

2.2.2 Links

http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml, Google's C++ coding style guide

http://www.boost.org/, online documentation and mailing lists

http://qt.nokia.com/doc/, online documentation

http://www.cmake.org/cmake/help/documentation.html, online documentation

http://stackoverflow.com/, online programming question and answer board

2.3 Project Layout

2.3.1 Project Files

We use CMake to generate Visual Studio projects and makefiles, because CMake enables cross-platform builds and programmatic project management. Thus, our CMake files, as the very definition of the OpenStudio project, are kept in the repository alongside the source files. Visual Studio projects and makefiles are generated files that should not be modified or checked in to the repository.

2.3.2 Source Files

Header and source files are kept in the same directory. File names must start with a letter, and cannot be all uppercase. File names follow UpperCamelCase convention, and can include digits, but not special characters (including `.`, `-`, and `_`). Also, unit tests are stored in a `test` directory immediately beneath the tested source code.

Files are named corresponding to the class that they contain, e.g. the class `Building` is located in `Building.hpp` and `Building.cpp`. Typically we limit ourselves to one class per file. However, small helper classes may be defined alongside a primary object, e.g. `BuildingParameters` could also be defined in `Building.hpp` and `Building.cpp`. Impl classes should be defined in separate files, e.g. `Building_Impl.hpp` and `Building_Impl.cpp`, to signify that users should not access these classes directly.

We use the following file extensions to differentiate between file types as well as whether or not the file is autogenerated. Autogenerated files should not be modified by hand as they are rebuilt automatically during the build process. Ensure that autogenerated files are deleted when the clean target is built and that they are generated in the build rather than the source directory. Additionally, unit tests and test fixtures end in `_GTest.hpp`, `_GTest.cpp`, `Fixture.hpp`, or `Fixture.cpp` respectively so they may be excluded from code coverage analysis.

1. C header files — .h
2. C source files — .c
3. C++ header files — .hpp
4. C++ source files — .cpp

5. Swig interface files — .i
6. Autogenerated header (C or C++) files (do not edit) — .hxx
7. Autogenerated source (C or C++) files (do not edit) — .cxx
8. Autogenerated Swig interface files (do not edit) — .ixx
9. Unit test files — *_GTest.hpp or *_GTest.cpp
10. Unit test fixtures — *Fixture.hpp or *Fixture.cpp

2.3.3 External Dependencies

Changing dependencies imposes a significant burden on the project, therefore decisions to add or remove dependencies must be discussed by the entire team. To add a new dependency the following conditions must be met:

1. Dependency license passes legal review
2. Must be active open source project with commercial adoption
3. Need for new functionality outside of current dependencies must be justified
4. Must be approved by majority of NREL OpenStudio team

Our current list of library dependencies, each of which meets these qualifications is:

1. boost
2. boost-log
3. Qt
4. Qwt
5. GTest
6. CLIPS
7. Expat
8. libssh
9. openssl
10. litesql

Our current list of executable dependencies, each of which meets these qualifications is:

1. CMake

2. EnergyPlus
3. Radiance
4. SWIG
5. Ruby
6. Doxygen
7. Graphviz
8. DAKOTA

When selecting classes or items from a library to do a specific task we prefer to use more general libraries and tools first as they place fewer restrictions on the code using them. If the functionality does not already exist in OpenStudio we prefer to use, in order, the C++ Standard Template Library, boost, Qt. Certain dependencies may not be required based on conditional configuration variables. Small libraries or tools may be included directly in the main OpenStudio project and built using CMake. Large libraries or tools should be installed separately. This reduces the start up cost of becoming a new developer.

2.4 Naming

2.4.1 Namespaces

The top level namespace for OpenStudio code will be `openstudio`. All classes and functions should be placed in the `openstudio` namespace. All macro definitions are written in the global namespace.

Namespaces are all lowercase so they can be easily distinguished from classes. For instance, `openstudio::energyplus::ReverseTranslator` indicates the `ReverseTranslator` class located in the namespace `openstudio::energyplus`.

Namespace structure includes `openstudio` plus the first level of source code hierarchy, except that utilities is not used as a namespace (`openstudio::IdfFile`, not `openstudio::utilities::IdfFile` or `openstudio::utilities::idf:` because classes in utilities are used widely throughout OpenStudio.

Use the `detail` namespace to hide classes or functions which are not meant to be part of the public interface (e.g. `openstudio::idd::detail::IddObject_Impl`).

2.4.2 Other Naming Conventions

- Classes are upper camel case, e.g. `DetailedGeometry`
- Member functions and local variables are lower camel case, e.g. `addVertex()` and `timeOfDay`
- Private member variables are lower camel case prefixed with `m_`, e.g. `m_name`. Private member functions are not prefixed with `m_`, member variables which are function pointers are prefixed by `m_`.
- Implementation classes using the pImpl idiom are postfixed by `_Impl`, e.g. `IddFile_Impl`.

- Enumerations and enumerated items are upper camel case, e.g. `enum Items{SmallItem, BigItem}` not `enum Items{smallItem, bigItem}`
- Unit test cases and fixtures are upper camel case. To ensure that test cases are easily recognizable in standard output, use `_` to indicate spaces, e.g. `TEST_F(SolarSystemFixture,Moon_Constructor)`.
- For "getter" type methods (e.g. to get a name) we prefer `object.name()` to `object.name` or `object.getName()`, unless the method accepts an argument. In that case we prefer a `get` prefix to clarify the verb associated with the argument(s), for example `object.getVariable(const String& varName)` is preferred to `object.variable(const String& varName)`. The get prefix applies even when the argument is an index into an array. For example `object.foos()`, but `object.getFoo(unsigned index)` is preferred to `object.foo(unsigned index)`.
- For "setter" type methods (e.g. to set a name) we prefer `object.setName(const String& name)` to `object.name(const String& name)`.

2.4.3 Naming Descriptiveness

Names for namespaces, classes, variables, and functions should be sufficiently descriptive that they convey their meaning and typical usage. We prefer to spell out words rather than use abbreviations to maintain clarity and readability. Again, readable and comprehensive names are preferred over names that are short and easy to type. Names should be as specific to the context as possible, e.g. Coordinate rather than Number. Names with global scope should be distinctive and unique (however, global scope should be avoided when possible).

2.5 Files

2.5.1 Include Statements

Reference files in your include statements from the top level OpenStudio source directory. For example, `#include <building/Building.hpp>` rather than `#include <Building.hpp>` or `#include "Building.hpp"`.

List include files in the following order:

1. Header files being implemented in this .cpp file.
2. Header files from the same project.
3. Other OpenStudio header files.
4. External headers files, grouped by namespace (boost, std, etc.).

We prefer to surround include files with angle brackets rather than quotes.

2.5.2 Using Statements and Namespace Aliases

Do not

1. use any using directives in any header file at the file level.
2. use any namespace aliases in any header file at the file level.
3. use any using directives to expose all names in a namespace, e.g. 'using namespace std'.

You may

1. use a class- or function-specific using declaration anywhere in a source file, e.g. 'using std::vector'.
2. use a class- or function-specific using declaration in functions, methods, or classes in .hpp files (below file scope).
3. use a namespace alias anywhere in a source file, e.g. 'namespace building = openstudio::building'.
4. use a namespace alias in functions, methods or classes in .hpp files (below file scope).

2.5.2.1 Include Guards

Every header file must contain a "guard" that prevents it from being multiply defined. Traditional include guards are thought to be more portable than #pragma once syntax. Place an include guard macro at the top and bottom of every .hpp file. The include guard defines a symbol which mirrors the entire file path from the source directory, ensuring that it will be unique across the entire project.

```
#ifndef UTILITIES_CORE_STRING_HPP
#define UTILITIES_CORE_STRING_HPP
// ...
#endif //UTILITIES_CORE_STRING_HPP
```

2.5.2.2 Macros

The use of macros is discouraged as the same effect can often be achieved with another method that is easier to debug and maintain. Any macros deemed necessary must be well commented to help with debugging and maintenance burden. If there are parameters, then the comment must describe them, including their type. The comment must describe what the macro is supposed to do. Example usage should be provided to clarify the intended use. If there are any side effects associated with the macro (such as incrementing a global variable, the need for certain local identifiers to be defined, etc), then list them.

2.6 Classes

2.6.1 Object Oriented Design

We encourage the use of Object Oriented (OO) design because it helps encapsulate behavior and logic in individual classes. Care should be taken to limit the coupling between classes so that they can be used in more general ways than originally anticipated.

2.6.2 Class Header Files

The following is required in class header files.

1. Surround code with an include guard (Section 2.5.2.1).
2. Forward declarations of other classes are preferred over #including the header files of those classes. #includes are only required if the class is actually used in the declaration, not if it is merely referenced. For instance, an #include is required if an instance of the other class is a member variable of the class being declared, but not if a pointer (including `boost::shared_ptr`) to that class is a member variable. A forward declaration is also sufficient for classes referenced in member function declarations.
3. Classes in the public API must include Doxygen documentation. Otherwise, practice good commenting habits, and use our code review comment format when appropriate (Section 2.10.2).
4. Use `const` appropriately (Section 2.1, Section 2.7.5).
5. Place typedef statements and non-member helper functions related to the class immediately below the class declaration.
6. Mark functions defined in header as inline.
7. Refrain from using convenience typedef names in public header files. For instance, `boost::optional<std::string>` is preferred to `OptionalString` (in public hpp only). While the latter is more readable (and shorter to type), the longhand version helps SWIG and Doxygen.
8. #include the appropriate `*API.hpp` file and place the appropriate `*_API` macro call between the `class` keyword and your class name if your class is to be included in a OpenStudio dynamic library. (See the utilities project for an example.)

2.6.3 Class Source Files

The following is required in class source files.

1. Source files should #include header files instead of using extern declarations.
2. Practice good commenting habits, and use our code review comment format when appropriate. Doxygen comments do not need to be written or maintained for cpp files (Section 2.10.2).
3. Follow the #include order outlined in Section 2.5.1.
4. Try to keep definitions (member and non-member functions, data initializations) in the same order as the declarations in the hpp file.

2.6.4 Class Definitions

- The protection of every base class should be specifically stated. When one class is derived from another, the default protection is private. Example:

```
class D : B {}; // D is privately derived from B
```

This is counter-intuitive as public inheritance is more commonly used. Thus, the compiler will generate a warning for unspecified protection levels. It is best to specifically state whether the inheritance is public, protected, or private. Example:

```
class D : public B {}; // correct
```

- Most designs require only public inheritance, any exceptions to this should be thoroughly documented.
- Class organization: to provide a common layout of the interface a class offers, the definition of the class should contain the following, in this order:
  1. declaration of any friend functions and classes
  2. the "public:" specifier
  3. declarations of any static public functions
  4. declarations of any static public data members
  5. declarations of any public member functions
  6. declarations of any public data members
  7. the "protected:" specifier
  8. declarations of any static protected functions
  9. declarations of any static protected data members
  10. declarations of any protected member functions
  11. declarations of any protected data members
  12. the "private:" specifier
  13. declarations of any static private functions
  14. declarations of any static private data members
  15. declarations of any private member functions
  16. declarations of any private data members

Notes:

- protected and private data member names should start with "m_"
- the return type of a function must be explicitly stated
- Constructors, destructors, virtual functions, and input/output functions should not be inline

2.6.5 Code Definitions in Header Files

1. Any code defined in a header file should be marked inline or be a template specification. Inline functions should be used only for single line get, set, and add methods.
2. Constructors completely defined by initialization commands may also be present in header files.

2.6.6 Inheritance and Virtual Functions

1. Every class with one or more virtual functions should have a virtual destructor. Since a derived class object may be deleted through a base class pointer, the base class must have a virtual destructor to ensure that the derived class destructor gets called. Otherwise, only the base class destructor will be called, resulting in a probable memory leak.
2. A derived class should not redefine a default parameter for a virtual function.
3. A derived class should not redefine an inherited non-virtual function.

2.6.7 Friends

1. Since friends violate encapsulation and information hiding, they should be avoided whenever accessor functions can be used just as efficiently.
2. Classes that require some amount of friendship with other classes should consider whether the scope of the friendship can be limited to certain methods.

2.6.8 Nested Classes

Nested classes other than enumerations are not supported by SWIG and should not be used in exported classes.

2.7 Functions

2.7.1 Inline

Use of `inline` is discouraged as it increases compile time. Only use `inline` if profiling indicates that a major speed up would result.

Details

Inline is a function specifier that hints to the compiler that a function should be expanded inline at compile time (C++ In a Nutshell pg 307). The compiler is free to ignore the directive. If the compiler does honor the hint, it generates code that is more tightly coupled and larger (Meyers (2005) #30). Also, inlining by default falls into the category premature optimization (http://en.wikiquote.org/wiki/Donald_Knuth#Computer_Programming_as_an_Art_.281974.29). It is easy for a profiler to tell us what functions should be inlined, but impossible for it to tell us which ones we inlined but should not have (C++ Coding Standards #8). Finally, any modern compiler will inline functions for you as part of its optimizations (http://msdn.microsoft.com/en-us/library/47238hez.aspx, http://gcc.gnu.org/onlinedocs/gcc-4.4.2/gcc/Optimize-Options.html).

SWIG

The tighter coupling that is created by using inline may limit the ability to provide your C++ library implementation in a library that can be replaced. That is, if inlining is used, you may have to recompile the

SWIG wrapper library for each change made to the corresponding C++ library. Therefore, it is strongly recommended that `inline` not be used when generating SWIG libraries.

2.7.2 Function Overloading

Except for constructors, function overloading should be reserved for functions in which the meaning of the overload is clear. Overloading and calling overloaded functions can decrease readability because it may not be clear to the reader which version of the function is being called. Functions should not be overloaded on the const-ness of the input arguments unless absolutely necessary and thoroughly documented.

2.7.3 Passing arguments

1. Pass double, int, unsigned, and bool as such, no need for const modifiers.
2. Pass all other function arguments as const& whenever possible to avoid unnecessary copying.

2.7.4 Return values

When returning multiple values from functions we prefer to return an object or struct. Alternatively, pair or tuple types may be returned provided that the meaning of each member of return type is well documented. We do not suggest returning multiple values by passing in reference arguments. Return values from functions should follow these examples:

```
// Return a plain data type.
double doubleFunc(double x){
  double result = 2.0*x;
  return result;
}

// Return a plain data type that may or may not be set.
// Caller must assume it may not be set, similar to a pointer return value
boost::optional<double> doubleFunc2(double x){
  boost::optional<double> result;
  if (x >= 0){
    result = 2.0*x;
  }
  return result;
}

// The following examples are related to returning references
// to a vector stored in a class
class BigObject{
 public:

  // return a const reference to the vector, consider this a const method
  // it may be better to return a copy and not a reference for multi-threading
  // applications
```

```
  const std::vector<Object>& objects() const {return m_objects;}

 private:

  std::vector<Object> m_objects;
};
```

2.7.5 Const-Correctness

Use const whenever possible (Meyers 2005; Sutter and Alexandrescu 2004; **?**).

Details

Effective C++ suggests using const for any value that should never be modified, to provide compile-time guarantees against logic errors. Example:

```
    void foo(const std::vector<int> &v)
    {
        const size_t size = v.size(); // Cache size in const value
        ... // use size_t as a cached value
    }
```

Similarly, any methods that should not modify object data, such as "getter" methods should be declared const:

```
    class MyType
    {
        public:
            int getValue() const
            {
                return m_value;
            }
        private:
            int m_value;
    };
```

Getter methods on custom containers that return references to internal data need to provide both const and non-const versions to ensure that the container can be used in as many contexts as possible.

Standard containers such as std::vector are implemented in this way.

```
    class Container
    {
        ...
        Type &operator[](int index) { ... }
        const Type &operator[](int index) const { ... }
        ...
    };
```

Exceptions

Do not provide both const and non-const overloads for parameters. The subtleties for when one version would be chosen over another are nuanced, and the meaning of the function is likely unclear:

```
// Big don't: Functions overloaded by argument constness
void doSomething(const std::vector<int> &p);
void doSomething(std::vector<int> &p);
```

Similarly, do not implement both const and non-const versions of templated types, unless the design specifically needs it. The danger is that the number of permutations would grow too quickly and the meaning of the code being called is likely lost and the implementation is probably duplicated:

```
// Similar don't (unless absolutely necessary)
void doSomething(const std::vector<boost::shared_ptr<int> > &p);
void doSomething(const std::vector<boost::shared_ptr<const int> > &p);
```

Also:

```
// Unnecessary except in the most strictly defined code, instead ...
std::vector<boost::shared_ptr<int> > Class::getData();
std::vector<boost::shared_ptr<const int> > Class::getData() const;
// Choose this:
std::vector<boost::shared_ptr<int> > Class::getData() const;
```

SWIG

Even though we know that SWIG discards const qualifiers (http://www.swig.org/Doc1.3/SWIGPlus.html#SWIGPlus_nn37), we should not let that modify our use of const. Our exposed C++ will rarely, if ever, live in isolation. We want to make the most use of the C++ type system; appropriate application of const can help us find logic errors in our code at compile time (Sutter and Alexandrescu 2004).

We should be aware (and document where possible) that if both const and non-const versions of a function exist the const version will likely never be called from within our SWIG code.

2.8 General

2.8.1 Strings

We are using std::string to store all character data. We assume that all std::string objects are UTF-8 encoded. String conversion functions are provided `utilities/core/String.hpp`. Other types of strings (such as QString or std::wstring) may be used to interface with users through GUI, command line, or SWIG bindings, but should be converted to std::string as soon as possible within the OpenStudio code base.

2.8.2 Paths

Use the typedef `path` in `utilities/core/Path.hpp` for all operations which access the filesystem. This will be `boost::filesystem::wpath` on Windows and `boost::filesystem::path` on POSIX. Use the file

stream classes derived from `boost::filesystem::fstream` rather than `std::fstream` as they can take either version of `path`.

2.8.3 Typedefs

Use typedefs to shorten the name of commonly used templates. In order to provide a consistent look and feel `std::vector`'s of objects should be suffixed with `Vector` (e.g. `IddKeyVector`), `boost::shared_ptr` should be suffixed with `Ptr` (e.g. `IddKeyPtr`), and `boost::optional` prefixed with `Optional` (e.g. `OptionalIddKey`). These names will correspond to the class names given to template instantiations exported by SWIG.

Exception: Use of these convenience typedefs is encouraged in all settings **EXCEPT** in public API header files, where we prefer types fully spelled out, as this helps Doxygen and SWIG.

2.8.4 Memory Management

Avoid using malloc, free, or delete. Use smart pointers instead, and only call `new` within a smart pointer constructor. This leaves memory management to the smart pointer and helps avoid crashes and memory leaks.

2.8.5 Avoid Code Duplication

Refactor code to avoid duplication wherever possible.

2.8.6 Flow Control

All control structures must have braces surrounding the body of the code within the structure, even if the body is only one statement, or even no statements. The body must be indented relative to the control statement.

All switch statements must have a default case at the bottom even if its only purpose is to report an error. In C++, the switch statement must be manually exited with a break statement, otherwise execution continues to fall through. Usually, falling through is undesirable behavior. If falling through is actually desired for a particular switch statement, that case must be commented well to explain why falling through is required.

2.8.7 Serialization

The boost Serialization library requires that classes register all member variables to be serialized. This is a potential source of error as variables may be added to the class but not to the serialization. Where possible serialization should be tested in unit tests.

Additional issues arise when serializing objects derived from an abstract base class. In this case, the abstract base class should register itself with the macro `BOOST_SERIALIZATION_ASSUME_ABSTRACT` (in the class header file); the derived classes should serialize their base object explicitly using the macro `BOOST_SERIALIZATION_BASE_OBJECT_NVP`, and should be registered in an associated source file, for example:

```
#ifndef __APPLE__
BOOST_CLASS_EXPORT(openstudio::SectionHeading);
BOOST_CLASS_EXPORT(openstudio::Text);
BOOST_CLASS_EXPORT(openstudio::Table);
#endif
```

is an excerpt from `src/utilities/document/Document.cpp`.

2.8.8 SWIG Support

All C++ types parsed by SWIG should have a definition available to SWIG. This will produce the most effective and usable target language wrappers. To accomplish this goal, a few things should be kept mind.

Exported templates

Template classes must be explicitly instantiated and exported in SWIG. In general, specific uses of a template class that are common enough to have a typedef should be exported. The same name as the typedef should be used for consistency.

Limit the Number of Types Exposed

The SWIG maintenance burden is primarily driven by the number of types that are exposed. Therefore, be judicious in choosing types to export.

Be Careful about STL Types

The STL should be relied on as heavily as possible for yielding portable and reliable code. However, not all STL types are supported by all SWIG target languages, so the STL types utilized should be carefully chosen. A reading of the .i files provided for each target language in the SWIG distribution (http://swig.svn.sourceforge.net/viewvc/swig/trunk/Lib/ruby/, http://swig.svn.sourceforge.net/viewvc/swig/trunk/Lib/csharp/) provides the most reliable and accurate information about which types have shipped support. Choose the minimum set of all types that are supported across all languages you want to target.

All STL container types support easy conversions between themselves via iterators. Therefore, it is possible that using std::deque internal to the application is the best choice while having accessors that return std::vector.

```
std::vector<MyType> get_data() {
  return std::vector<MyType>(m_deque.begin(), m_deque.end());
}
```

Use the Pimpl Idiom

A common C++ technique known as the pImpl Idiom can be used to provide a streamlined interface to a complex, internal class (http://c2.com/cgi/wiki?PimplIdiom, Sutter and Alexandrescu (2004) #43). This technique is preferred for OpenStudio classes that require both SWIG export and features primarily supported by pointers (polymorphism, data sharing).

2.9 Code Portability

The following simple rules will help code move more easily from one target platform to another. Word lengths, byte and word order, alignment, argument evaluation, signs, and sign extensions are the major causes of portability problems. To avoid such issues:

1. Avoid casting variables.
2. Avoid assumptions about signed variables (especially characters). Be very careful when mixing signed and unsigned variables and when using the modulus operator.
3. Use standardized functions, macros, and typedefs porting inconsistencies.
4. Avoid code that depends on the evaluation order of function arguments.
5. All characters that are used in non-arithmetic cases (bit fields, etc.) must be declared to be unsigned.

Intentional non-portable code should be well documented and separated from portable code. For example, if a certain piece of functionality requires several lines of non-portable code, then the developer should put the non-portable code in a separate function, method or macro and invoke it from the primary function.

Standard libraries

Prefer the C++ standard libraries unless operating system specific calls are required. Place all operating system specific implementations in one or more classes that can be easily replaced at compile time, centralizing all of your OS specific code.

Boost

Boost is a widely used, highly portable C++ library. It is often used as a staging ground for up-coming C++ language and standard library features. The use of Boost libraries is generally allowed, and is preferred to the C++ standard libraries in some cases: boost::filesystem, boost::function and boost::asio.

The use of boost is not used recommended for SWIG facing code, with the exceptions of boost::shared_ptr and boost::optional.

Multiple Compilers

Developers are encouraged to use multiple compilers to ensure portability. Furthermore, an extra level of confidence can be gained by knowing that your code generates few or no warnings at high levels across multiple platforms.

C++0x / TR1

The next version of the C++ language standard, C++0x, is under development. Most current compilers support experimental library features inside the TR1 namespace.

We are currently not recommending the use of TR1 library features, since the language specification is still in flux. Instead, use the Boost library, which is more stable and portable.

2.9.1 Compiler Warnings

Compiler warnings should be enabled to the highest "normal" levels. Compiler warnings should be eliminated before code release.

Details

All compilers support esoteric warnings that generate more noise then useful information. However, all warnings that the compiler authors think are important should be enabled. They will often help you find portability and logic problems. For example, in Visual C++ at least warning level 3 (/W3) should be enabled, as it is recommended for production code (http://msdn.microsoft.com/en-us/library/thxezb7y.aspx). Similarly, -Wall warnings should be enabled in G++ (http://gcc.gnu.org/onlinedocs/gcc-4.4.2/gcc/Warning-Options.html). An interesting option for G++ that is not enabled with -Wall is -Weffc++, which generates warnings for programming standards specified in Meyers (2005).

2.9.2 Exceptions

We allow our code to throw exceptions. However, the number of exceptions thrown should be limited to the absolute minimum required. Exceptions should not be used to pass expected information between components and should not be used as "convienient" methods for changing program execution paths. An exception means something very bad has happened and the code cannot continue on in the manner it was called. All thrown exceptions must be logged, see 2.9.3; use of the `LOG_AND_THROW` or `LOG_FREE_AND_THROW` macros is encouraged. Valid situations to throw exceptions include object constructors called with parameters that would invalidate an object's state (e.g. conversion from string to number fails) or failed attempts to access resources (e.g. missing files or out of memory errors). All exceptions thrown in our code must be derived from `std::exception`, and should not be identically equal to `std::exception` (because gcc does not recognize `std::exception(const char*)`). Additionally, all methods that throw exceptions must be documented as such; it is assumed that unmarked methods do not throw. Because the C++ language does not provide a method to enforce the types of exceptions thrown at compile time we mark exception throwing methods with comments like `/** throws ExceptionType1, ExceptionType2, ... */`.

Function calls to other libraries that may throw exceptions under relatively normal circumstances (`boost::filesystem` comes to mind) must be wrapped in a try/catch block. If an exception is caught it may be rethrown following the above guidelines.

2.9.3 Logging

Log messages alert users of unexpected conditions and assist developers with debugging. Several convenient logging macros are provided. All classes should use the macro `REGISTER_LOGGER("my.log.channel")` within their private scope to register a logging channel associated with the class. The log channel name is hierarchical with levels separated by '.'. Each object should scope its log to its namespace and class name. For instance the class `openstudio::model::Building` should use channel `openstudio.model.Building`. The following logging levels are available:

1. Debug - detailed information for developers to use in debugging
2. Info - similar to Debug, but at a higher level, and possibly of interest to super-users
3. Warn - a situation which is likely the result of bad input or will result in suspect output and should

be brought to the attention of the user

4. Error - a situation which is definitely the result of bad input, will result in bad output, and must be brought to the attention of the user
5. Fatal - the program cannot continue

Logging macros are also provided for Free functions. However, there is no `REGISTER_LOGGER` macro for free functions so they must use the macros `LOG_FREE`, which requires an additional argument to specify the logging channel. The channel should be the namespace and then function name. So function `int openstudio::model::foo()` would log to `openstudio.model.foo`. The following example shows how to use the logging macros:

```
namespace foo{

  // A class with logging
  class Bar{
  public:
    Bar(){}
    void doSomething();
  private:
    REGISTER_LOGGER(``foo.Bar");
  };

  // using logging in a member of a class with logging
  Bar::doSomething()
  {
    // appropriate debug statement
    LOG(Debug, ``Entering Bar::doSomething");

    bool trouble = isTrouble();
    if (trouble){
      // appropriate error statement
      LOG(Error, ``Ran in to trouble");
    }

    // using stream operators in a log statement
    LOG(Debug, ``Leaving Bar::doSomething, trouble = " << trouble);
  }

  // using logging in a free function
  bool isTrouble()
  {
    int a = 1;
    int b = 1;
    int sum = a + b;

    // first argument is the channel, second is the log message
    LOG_FREE(Debug, ``foo.isTrouble", ``Entering foo::isTrouble");

    bool result = false;
```

```
    if ((a != b) || (sum != 2)){
        LOG_FREE(Error, ``foo.isTrouble", ``Trouble detected, a = " << a <<
                        ``, b = " << b << ``, sum = " << sum);
        result = true;
    }

    LOG_FREE(Debug, ``foo.isTrouble", ``Leaving foo::isTrouble, " <<
                    ``result = " << result);
    return result;
  }
```

2.9.4 Unit Tests

We aim to sufficiently test all OpenStudio code to ensure that it functions as expected, both in regular use and for corner cases. Writing unit tests before the code is fully functional is highly recommended. When fixing bugs or tracing down input specific errors we ask that those problem cases be added to the unit test suit to trap future error conditions or similar bugs. We compute code coverage analysis as one indicator of whether or not code is tested sufficiently.

An important convention is that no output be written to standard out or standard error in unit tests. Diagnostic information may be logged using the logging macros described in 2.9.3. Each test fixture disables logging output to standard out and sends it to a file named `test_fixture_name.log`.

2.10 Formatting

2.10.1 Indentation and Wrapping

We standardize on the following to improve code readability:

1. Wrap lines at 100 columns. If a line needs to be split, then the split should occur at some functional location (e.g., before/after the operator) and not just after the 100th character.
2. Indentation level/tab character is two spaces.
3. Use Windows-style line endings.
4. Indentation rules:

```
#ifndef BUILDING_MODEL_BUILDING_H // no indent
#define BUILDING_MODEL_BUILDING_H

namespace building{ // no indent
namespace model{ // no indent

class Building{ // no indent
 public: // 1 space
```

```
  Building(){} // 2 spaces

  openstudio::String name() const; // 2 spaces

 private: // 1 space

  openstudio::String m_name; // 2 spaces
};

// close each namespace on its own line, comment with name of namespace
} // model
} // building

// close the include guard
#endif // BUILDING_MODEL_BUILDING_H
```

Functions declarations and calls with many arguments can become too long to fit on a single line, in which case we recommend:

```
// Preferred
ReturnType ClassName::FunctionName(Type par_name1, Type par_name2) {
  DoSomething(); // 2 space indent
  ...
}

// If there is too much text to fit on one line:
ReturnType ClassName::ReallyLongFunctionName(Type par_name1,
                                             Type par_name2,
                                             Type par_name3) {
  DoSomething(); // 2 space indent
  ...
}

// If you cannot fit even the first parameter:
ReturnType LongClassName::ReallyReallyReallyLongFunctionName(
    Type par_name1,  // 4 space indent
    Type par_name2,
    Type par_name3) {
  DoSomething();  // 2 space indent
  ...
}

// Const always goes on same line with closing ), last parameter, and {
ReturnType LongClassName::ReallyReallyReallyLongConstFunctionName(
    Type par_name1,  // 4 space indent
    Type par_name2,
    Type par_name3) const {
  DoSomething();  // 2 space indent
  ...
```

```
}
```

Detailed notes:

1. The return type is always on the same line as the function name.
2. The open parenthesis is always on the same line as the function name.
3. There is never a space between the function name and the open parenthesis.
4. The open curly brace is always at the end of the same line as the last parameter.
5. The closing curly brace is either on the last line by itself or (if other style rules permit) on the same line as the open curly brace.
6. There should be a space between the close parenthesis and the open curly brace.
7. All parameters should be named, with identical names in the declaration and implementation.
8. All parameters should be aligned if possible.
9. Default indentation is 2 spaces.
10. Wrapped parameters have a 4 space indent.

2.10.2 Comments

Documentation is an important part of any software project. Even the best code is hard to understand or maintain if it is not properly commented. Developers are encouraged to insert as many comments as needed to explain the code.

2.10.2.1 Public Interface

Publicly available libraries, functions, classes, class methods, etc. are to be documented with Doxygen (http://www.stack.nl/~dimitri/doxygen/manual.html). To generate our Doxygen html documentation, build the corresponding `_doc` project(s), which appears when the `BUILD_DOCUMENTATION` option is checked in CMake. There is a link to each project's `index.html` in `trunk/doc/index.html`.

2.10.2.1.1 Classes and Free Functions

The following syntax is preferred for documenting classes, methods, and free functions:

```
/// Brief description. More descriptive text follows the first period.

/** Brief description. More descriptive text follows the first period. This
 *  format is a little more readable (in the code) when there are multiple
 *  lines */
```

Such descriptive code blocks should be placed directly above the item being documented, when it is declared (in its header file). Doxygen provides basic markup commands to help with formatting:

```
\code, \endcode
\link, \endlink
\li
<b></b>
<ol></ol>
<ul></ul>
```

Also see http://www.stack.nl/~dimitri/doxygen/commands.html, and http://www.stack.nl/~dimitri/doxygen/htmlcmds.html. Please use `\code, \endcode` to markup example usage blocks:

```
 * \code
 * IddObject::properties().memo
 * IddObject::properties().minFields
 * IddField::properties().autosizable
 * \endcode
```

and use `\link, \endlink` to write readable hyperlinked text. For instance,

```
 * ... for parsing and accessing \link IddFile IddFiles\endlink, and
 * \link IddObject IddObjects\endlink ...
```

becomes, " ... for parsing and accessing IddFiles and IddObjects ...," while still providing links to `IddFile` and `IddObject`.

To avoid needlessly verbose documentation, we prefer to document input arguments and return types in the descriptive text, rather than using `\param`. (Even without `\param` tags, Doxygen does a good job of highlighting the arguments (type and name) and return type, and our naming conventions should do a lot to suggest the meaning of input arguments without us providing a detailed description.) Instead, focus on documenting pre- and post-conditions, exceptional situations, and providing example usage.

`oOPENSTUDIO_ENUM` is an exceptional case where we use a macro to generate new classes. To ensure documentation of those classes, explicitly declare the class name, as in:

```
/** \class IddFileType
 *  \brief Enumeration of the IddFiles available through IddFactory. */
oOPENSTUDIO_ENUM( IddFileType,
  ((EnergyPlus))
  ((OpenStudio)) );
```

(Note that other Doxygen commands similar to `\class` are available, and may be of use in other situations where Doxygen has a hard time linking a documentation block with the item being documented. For instance, `\file` may be useful in some circumstances.)

A free function may be associated with a class by prepending its documentation with a `\relates` command. For example,

```
/** \relates WorkspaceObject
```

```
 *  Helper function to get the handles of a vector of objects */
UTILITIES_API std::vector<Handle> getHandles (const std::vector<WorkspaceObject>& objects);
```

causes the documentation for getHandles to show up on the WorkspaceObject (class) documentation page, rather than on the openstudio (namespace) page.

2.10.2.1.2 Libraries

Projects are documented by a special header file, `mainpage.hpp` that lives in that project's folder under `src`, and may be listed in the corresponding `CMakeLists.txt` file to make it easily editable through the developer environment. For large projects (like `OpenStudio_utilities`), there may also be use for one or more `page.hpp` files. Both types of files should primarily consist of one large code block surrounded by the namespace declaration of the project. For example:

```
namespace openstudio {
/** \mainpage OpenStudio Utilities
 *
 *  The utilities library provides common functionality used throughout  OpenStudio.
 *  OpenStudio's utilities live in the top level namespace of OpenStudio, openstudio.
 *  The following sub projects are defined:
 *  ...
 */
} // openstudio
```

(Placing the comment block in the namespace of the project being documented helps Doxygen link to related documentation. If there are items in a sub-namespace that need to be referred to, you can use the syntax `\link foo::Bar Bar\endlink`.) Mainpages show up as `index.html` in the generated HTML documentation, and are the top-most link in the navigation pane.

Following the above `mainpage.hpp` example, `page.hpp` files should start with a `\page` command, which differs in syntax from `\mainpage` in that it takes a label argument in addition to a title:

```
/** \page idd_page OpenStudio Idd
```

The reference tag (`idd_page` in the example) can then be used in that libary's documentation to provide a link to the page:

```
\ref idd_page
```

The `\section` command, which should be used in both `mainpage.hpp` and `page.hpp` files to break up the text into logical segments follows a similar convention, and its labels are also available through the `\ref` command.

`OPENSTUDIO_ENUM` and some other macros are called out in `trunk/Doxyfile.in` for Doxygen to expand in place whenever it is used. Consider adding your macro to this list if you are having trouble documenting the use of your macro in the public interface.

`_API` macros should be predefined in `trunk/Doxyfile.in` so they do not show up in the documentation. Search for `UTILITIES_API` in `trunk/Doxyfile.in` and follow its example.

2.10.2.2 Source Files

Source files should be well commented so that the intent of each step in the algorithm is clear. Every item defined in a header file must have a corresponding comment that describes it. The following general conventions should be followed when documenting protected and private code (i.e., `cpp` and the non-public portions of `hpp` files):

1. Use C++ style comments, i.e. the double-slash
2. There should be enough documentation in the file so that a person who is familiar with the programming language and generally familiar with the process can understand what the method is doing and how it is doing it.
3. Comments that are on the same line as the code must be spaced over to the right, to be easily distinguished from the code.
4. All variables except temporary loop index variables must be described.
5. All classes must be fully described where they are defined and should be briefly described wherever else they are used.
6. Interactions between classes should be described.

The comments in the code are obviously important documentation to keep up to date; they follow the source for the life of the code. When comments are kept accurate, the developers and others will benefit tremendously. Please take the time to review them and update them as necessary. Developer comments (e.g. from code reviews, etc) should be of the following format in order to retain information about who and when they were added "initials@date: comments". For example, "DLM@20091225: I don't like this code and that is why am I working on Christmas".

2.10.3 Old Code and Commented-out Code

There is no reason to keep old, unused code around since old code is always available through the svn repository. It only clutters the files and usually, after long periods, is not related to the surrounding "real" code. Therefore, old code must be removed. If there is some reason that some old code should be in the software base, but not executed, then the developer must comment out the code, with added explanatory comments. Code that is to be removed at a specific date or milestone (e.g. handling a special case in a soon-to-be deprecated version of a file) should be marked with the date or milestone at which it should be removed.

2.10.4 Temporary Code

There are times when a developer needs to put temporary code into the project. For example, a stubbed function, or an inefficient but quick implementation that only works for a small number of modules. Putting temporary code into the software base is discouraged, but if necessary it should adhere to these standards

so it may be easily identified and removed at a later date. The most common problem with temporary code is that, against intention, it becomes permanent. To overcome this problem, we require that all temporary code be marked with a comment block containing a unique string that can be easily located in the base. The string we will be using is:

```
// TEMPCODE START
// <describe why we have this code here, who is adding it (initials),
// date it was added>
... actual temp code block ...
// TEMPCODE END
```

If START/END does not make sense or is not practical for the temporary code block, at least the TEMPCODE markup.

3 Ruby Coding Standards

The Ruby bindings to OpenStudio include a mix of compiled Swig generated extensions, pure Ruby libraries, unit tests of both the extensions and Ruby libraries, and example scripts to help users get started. These elements are packaged together and included in the OpenStudio installer. Centralizing all OpenStudio functionality to a single install location allows user scripts to reference the OpenStudio API in a standard way across computers and projects. Users reference the OpenStudio API from project specific scripts that are out of the scope of the OpenStudio project. However, as user scripts become more developed and mature we encourage them to be added back into the official OpenStudio distribution either as example scripts (low complexity) or pure Ruby libraries (higher complexity). Ruby-only functionality that is widely used will be ported to C++ so it can be available in bindings to other languages as well as the core C++ tools. The Ruby coding standards generally follow the C++ standards. In this chapter we only describe points of departure or clarify places where the C++ coding standards conflict with normal Ruby conventions.

3.1 Project Layout

The following name conventions are in place for Ruby files:

1. Ruby file extension - .rb
2. Directory names - lowercase
3. File names (other than openstudio.rb) - UpperCamelCase
4. Ruby unit tests extension - *_Test.rb

To limit complexity of Ruby include paths, all Ruby tests and scripts are assumed to be part of the OpenStudio library, that the `lib` directory is in the user's Ruby path, and that all required shared libraries are in the system's path (this is achieved during the `require 'openstudio'` call). Therefore, all Ruby scripts and libraries should refer to each other relative to the `lib` directory.

3.2 Naming

The top level module for OpenStudio code is `OpenStudio`. Sub modules are also used and follow the C++ namespaces for Swig extensions. Other conventions are:

- Classes are upper camel case, e.g. `DetailedGeometry`
- Member functions and local variables are lower camel case, e.g. `addVertex()` and `timeOfDay`
- Enumerations as well as the enumerated items are upper camel case, e.g. `enum Items{SmallItem, BigItem}` not `enum Items{smallItem, bigItem}`
- Unit test cases are upper camel case and post-fixed with `\_Test`, individual tests are upper camel cased and prefixed by `test\_`. The idea is that test cases be easily recognizable in standard output when running a large number of tests, use `_` to indicate spaces for greater readability.

3.3 Unit Tests

We assume that C++ portions of OpenStudio are sufficiently tested using the C++ unit tests. Therefore, we do not require that all tests be duplicated for Ruby. However, we do encourage some testing of the Ruby extensions, particularly in areas where Swig does something novel to the exported classes (e.g. changes a name, maps types, etc). Pure Ruby libraries should be tested completely in Ruby. These tests are then ported to C++ unit tests when the functionality is added to the C++ code base.

3.4 Indentation and Wrapping

We standardize on the following formatting conventions to improve code readability:

1. Wrap lines at 100 columns. If a line needs to be split, then the split should occur at some functional location (e.g., before/after the operator) and not just after the 100th character.
2. Tab character is two spaces

3.5 Comments

Documentation is an important part of any source code. Even the most well written code is hard to understand or maintain if it is not properly commented. Comments should be made in RDoc format for documentation generation. The developer is encouraged to insert as many comments as needed to explain the code, not just the minimum required.